Dlaczego nie potrzebujesz wzorców projektowych w Pythonie?

...

Sebastian Buczyński @ wroc.py 2017-02

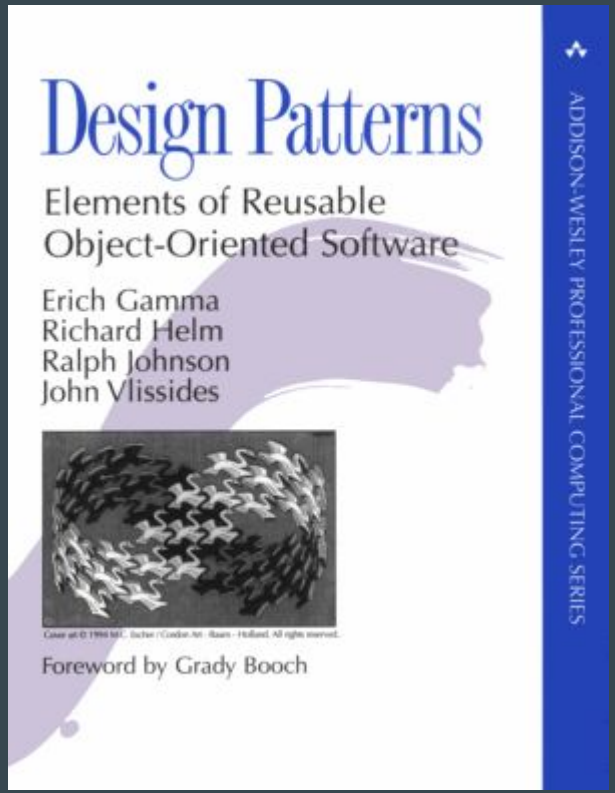
Kim jest ten koleś?!

STX NEXT
agile / programming / teams

@

Wzorce projektowe - co to?

Sprawdzone rozwiązanie
powtarzalnego problemu

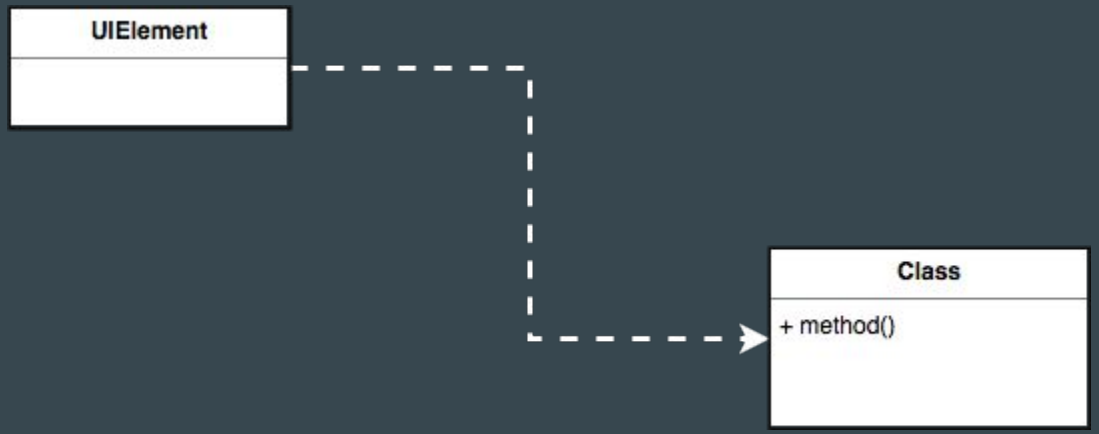


Wzorce projektowe są trochę jak frameworki webowe, tylko na innym poziomie

Żeby nie wynajdywać bez przerwy koła na nowo...

Wzorce projektowe - przykład

Komenda (Command)



Wzorce projektowe - przykład

Komenda (Command)

Wzorce projektowe - przykład

Komenda (Command)

Wzorce projektowe - po co to?

sformalizowanie *najlepszych praktyk* projektowania oprogramowania

Wzorce projektowe - po co to? 2

Lepsze komunikowanie zaimplementowanych rozwiązań

Wzorce projektowe - czy spełniają obietnice?

Częściowo.

Wzorce projektowe - czy spełniają obietnice? 2

Wciskanie wzorców na siłę tam, gdzie nie są potrzebne pogorszy sprawę, zamiast poprawić

Wzorce projektowe - czy spełniają obietnice? 3

Nowoczesne, dynamiczne języki mają wbudowane funkcjonalności, które ułatwiają implementację wzorców/eliminują kilka z nich

Wzorce projektowe, a Python

W książkach na ten temat pojawiają się implementacje 1:1 oryginalnych przykładów z *Design Patterns*, nie wykorzystując cech języka, które mogłyby uprościć wynikowy kod

Kilka wzorców prościej, bardziej pythonowo

- Singleton
- Command
- Iterator
- Dekorator

Singleton

W całym programie dokładnie jedna instancja danej klasy

Singleton - próba nr 1

```python
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super().__new__(cls, *args, **kwargs)
        return cls._instance
```

Singleton - próba nr 1

```python
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super().__new__(cls, *args, **kwargs)
        return cls._instance
```

```python
singleton1 = Singleton()
singleton2 = Singleton()
singleton1 is singleton2 # True
```

Singleton - próba nr 2

```
singleton = Singleton()

from not_existing_module import singleton
```

Singleton - próba nr 3

Moduł ma semantykę singletona.

Dokładnie jedna instancja, po załadowaniu istnieje w `sys.modules`

```python
# src/project/singleton.py

__all__ = ['some_fun']

def some_fun(arg):
    pass

module_var = None
```

Command - czy zawsze potrzebujemy klasy?


```
class Client:
    def some_fun(self):
        receiver = Receiver()
        command = Commmand(receiver)
        ui_element.set_command(command)
```

Command - czy zawsze potrzebujemy klasy? 2

```
class Client:
    def some_fun(self):
        receiver = Receiver()
        def command():
            receiver.do_some_action()
        ui_element.set_command(command)
```

Command - czy zawsze potrzebujemy klasy? 3

```
def some_fun(self):
    receiver = Receiver()
    command = functools.partial(receiver.do_some_action)
    ui_element.set_command(command)
```

Command - __call__ vs execute()

```
class SophisticatedCommand:

    def execute(self):
        result = receiver.do_some_action()
        another_receiver.do_even_more(result)

    def __call__(self):
        result = receiver.do_some_action()
        another_receiver.do_even_more(result)
```

Command - __call__ vs execute()

```
class SophisticatedCommand:

    def execute(self):
        result = receiver.do_some_action()
        another_receiver.do_even_more(result)

    def __call__(self):
        result = receiver.do_some_action()
        another_receiver.do_even_more(result)
```

```
sophisticated_command()
sophisticated_command.execute()
```

Iterator

Wbudowany w język. Żeby obiekt był iterowalny, wystarczy zaimplementować metodę __iter__, która zwróci iterator

```
class MusicBand:
    def __init__(self, guitarist, vocalist):
        self.guitarist = guitarist
        self.vocalist = vocalist

    def __iter__(self):
        return iter([self.guitarist, self.vocalist])

for musician in music_band:
    pass
```

Iterator - naturalne dla kolekcji, a poza tym…?

Implementacja __iter__ umożliwi “rozpakowanie” klasy do tupli

```
class Transaction:
    def __init__(self, buyer, items):
        self.buyer = buyer
        self.items = items

    def __iter__(self):
        return iter((self.buyer, self.items))

buyer, items = transaction
```

Własne, wyspecjalizowane iteratory

Własny iterator? Nic prostszego.

- kolejne wywołania __next__ mają zwracać kolejne elementy
- gdy nie ma już więcej elementów - __next__ rzuca StopIteration
- iterator też powinien być iterowalny - __iter__ może zwracać self

Iterate all the things!

```
>>> for message in p.listen():
...     # do something with the message
```

Alternatywa? Generatory

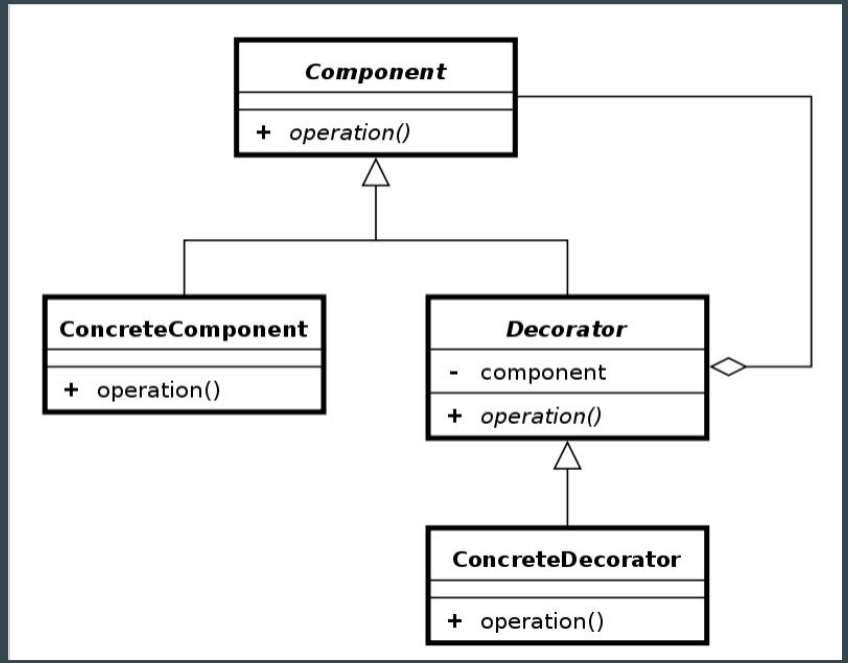
```
def listen(self):
    "Listen for messages on channels this client has been subscribed to"
    while self.subscribed:
        response = self.handle_message(self.parse_response(block=True))
        if response is not None:
            yield response
```

Dekorator

Rozszerzenie zachowania danego obiektu

Możliwe do zastosowania w czasie wykonania programu

Powinno się dać wielokrotnie dekorować ten sam obiekt, różnymi dekoratorami i w dowolnej kolejności



Dekorator - implementacja klasyczna

```
class Decorator:
    def __init__(self, decorated_obj):
        self.decorated_obj = decorated_obj

    @functools.lru_cache(maxsize=32)
    def get_data(self):
        self.decorated_obj.get_data()
```

I tak dla każdej metody.... także tych, których nie potrzebujemy wzbogacać :(

Dekorator - co się dzieje, gdy wołam metodę na obiekcie?

Metody są atrybutami, tak jak pola w klasie/obiekcie

Najpierw w ruch idzie __getattribute__, która sprawdza słowniki obiektu , a potem klasy (o ile nie korzystamy ze __slots__).

Jeżeli nie znaleziono atrybutu w typowych miejscach, to dalsze sterowanie jest przekazywane do metody __getattr__, która domyślnie rzuca wyjątek AttributeError

Dekorator - co się dzieje, gdy wołam metodę na obiekcie? 2

```
class ClsVsObjectDict:
    some_prop = 1
    def __init__(self):
        self.some_prop = 2

cls_vs_object_dict = ClsVsObjectDict()
cls_vs_object_dict.__dict__['some_prop']
cls_vs_object_dict.__class__.__dict__['some_prop']
```

Dekorator - minimalistyczna implementacja

```
class Decorator:
    def __init__(self, decorated_obj):
        self.decorated_obj = decorated_obj

    @functools.lru_cache(maxsize=32)
    def get_data(self):
        return self.decorated_obj.get_data()

    def __getattr__(self, name):
        return self.decorated_obj.__getattribute__(name)
```

Nie łamie kontraktu!

Czy to serio jest bardziej czytelne?

To zależy.

Nie można traktować wszystkiego jak gwóźdź, tylko dlatego że umie się akurat używać młotka

Readability counts.

Bibliografia

- Luciano Ramalho: *Fluent Python* http://shop.oreilly.com/product/0636920032519.do
- Tarek Ziadé: *Expert Python Programming* https://www.packtpub.com/application-development/expert-python-programming
- Chetan Giridhar: *Learning Python Design Patterns - Second Edition* https://www.packtpub.com/application-development/learning-python-design-patterns-second-edition
- Russ Olsen: *Design Patterns in Ruby* *http://designpatternsinruby.com/*
- E. Gamma, R. Helm, R. Johnson, J. M. Vlissides: *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku* http://helion.pl/ksiazki/wzorce-projektowe-elementy-oprogramowania-obiektowego-wielokrotnego-uzytku-erich-gamma-richard-helm-ralph-johnson-john-m,wzoele.htm
- David Beazley, Brian K. Jones: *Python. Receptury. Wydanie III* http://helion.pl/ksiazki/python-receptury-wydanie-iii-david-beazley-brian-k-jones,pytre3.htm
- Peter Norvig: *Design Patterns in Dynamic Languages* http://norvig.com/design-patterns/